The WideStudio is an Integrated Development Environment for Free and of Open Source, and also independent of encodings, platforms and operating systems. Since version 3.20, the WideStudio can deal with some script languages; Ruby, Python and Perl. You can develop applications using these scripts as well as C++.

This document introduce you how to use the script languages with the Application Builder.

WideStudio unofficial Tutorial — 2.
An Indroduction to Script Develompent

1 What's the WideStudio?

The WideStudio is an integrated development environment working on Linux, Windows, FreeBSD, Solaris and many other unix-like platforms. The project was started in 1999 by Mr. Hirabayashi, and the latest version 3.20 is released in Dec 2002.

1.1 Developing using scripts: MPFC

The libraries and the Application Builder of the WideStudio are written in C++, and you have to use also C++ for developing your application. Yes, so far.

From ver.3.20, the WideStudio supports **MPFC**, Multiple Platform Fundation Class; that is, the following languages are supported for a development of your application:

- Ruby (1.6 or later)
- Python (1.5.2 or later)
- Perl (5.6 or later)

Scritps that you'll write use the shared/dynamic libraries of the WideStudio, which is the same as what you have used with C++.

Therefore, of cource, you can still use C++ as before. If you don't use the scripts to develop applications, ok, skip this document.

1.2 What you can do?

By the new support of the scripts, an application can be composed of script files instead an exectable file made from C++.

Note that the script files use theire own shared/dynamic libraries$^{(1)}$ to cooperate with the shared/dynamic class libraries$^{(2)}$ of the WideStudio. The first ones$^{(1)}$ are copied to the directory where the script files are, while the second ones$^{(2)}$ are in a system directory such as **`/usr/local/ws/lib`** or **`C:\Windows\System32\`**.

1.3 How scripts use WideStudio?

Thnaks to SWIG (**`www.swig.org`**), the C++ class libraries of the WideStudio can be used from the scripts. If you have something not described here, to see documents on SWIG may help you.

2 Installation

2.1 Requirement

In order to read this document, at least you have to:

- use the WideStudio at least one time,
- know how to write a code in any of the script.

2.2 System requirement

If your system is Windows:

- Nothing. The scripts will be installed on the installation of the WideStudio.

If your system is an unix:

- Ruby have to be installed. The latest version of ruby is 1.6.7 (2002/Dec).
- Python have to be installed. The latest version of python is 2.2.2 (2002/Dec), but at least 1.5.2 is enough.
- Perl have to be installed. The latest version of perl is 5.8.0 (2002/Dec), but at least 5.6.0 is enough.

The scripts above are not necessary to be installed unless you want to use. For example, if you will use only a python script, you don't need ruby nor perl.

2.3 Installation on Windows

Unpack the zip archive of the WideStudio named "wswin320.zip". Then the folder "wsinst" will be created, and click "setup.exe" to launch the installer.

Proceed the installation dialog. After several clicking the OK button, you will find a confirmation text and check boxes to ask whether the scripts will be installed by the WideStudio installer.

I should note that even if you have already installed the scripts on Windows, you must install the scripts using the WideStudio installer because of the binary compatibility of DLLs. The scripts will be installed into the same folder with the WideStudio: typically C:/Program Files/WideStudio/.

After the installation, you need to reboot your computer for updating PATH and othe environment variables.

2.4 Installation on Unix

Follow as:

```
% tar xzvf ws-v3.20-src.tar.gz
% cd ws/src
% ./configure
```

then, you will see a messages like as follows:

```
OS: Linux
c++ compiler: g++
Has OpenGL
Has Jpeglib
Has Xpmlib
Has Pnglib
```

```
No ODBC library
No PostgreSQL development library
No MySQL development library
Has Python..
Has Ruby..
Has Perl..
```

If you have already the scripts, they have to be detected automatically. Otherwise, check your system (e.g., path to the script).

Then,

```
% make
% su
% make install
```

and also you have to set some environment variables described in other document of the WideStudio.

2.5 Setting properties

At the first time to use the scripts with the Application Builder, some project settings are needed.

Select the menu bar "Option" — "Environments". There are fields to be filled with a full path to the exectable of each script.

The following is an example:

Ruby `/usr/bin/ruby`

Ruby(DBG) `/usr/bin/ruby -d`

Python `/usr/local/bin/python`

Python(DBG) `/usr/local/bin/python -dv`

Perl `/usr/local/bin/perl`

Perl(DBG) `/usr/local/bin/perl -dv`

Make sure that the scripts with fullpathes specified above are exactly same with ones that you used to compile and build the WideStudio itself.

Note that, on Windows, the fields have been filled at the installation.

3 Building examples

3.1 Quick start

1. See the directories containing sample projects:

   For example,

   ```
   ws/samples/Ruby/hello/:
   btnop.rb        newproject.prj  newwin000.win
   newproject.col  newproject.wns
   ```

   or

   ```
   ws/samples/Python/hello/:
   btnop.py        newproject.prj  newwin000.win
   newproject.col  newproject.wns
   ```

2. Open the project file "newproject.prj" with the Application Builder.

3. Press “Execute” icon, or select “Build” — “Execute”. No need to build anything.
4. A window comes up to the screen. Pressing the “test” button, the string changes to “hello”.

After the execution, some files that are needed to run the scripts are copied to the directories. Then the directories have to have the following files:

```
ws/samples/Ruby/hello/:
btnop.rb  newproject.col  newproject.rb   newwin000.rb
mpfc.so   newproject.prj  newproject.wns  newwin000.win
```

or

```
ws/samples/Python/hello/:
_mpfcmodule.so   mpfc.pyc        newproject.wns
btnop.py         newproject.col  newwin000.py
btnop.pyc        newproject.prj  newwin000.pyc
mpfc.py          newproject.py   newwin000.win
```

3.2 How does it works

To make the “test” button turn to “hello”, just one script file is enough.

For ruby,

```
ws/samples/Ruby/hello/btnop.rb

def btnop(object)
        object.setProperty("labelString","hello");
end
```

For python,

```
ws/samples/Python/hello/btnop.py

import mpfc
def btnop(object):
        object.setProperty("labelString","hello")
        return
mpfc.WSGFfunctionRegister("btnop",btnop)
```

For perl,

```
ws/samples/Perl/hello/btnop.pl

sub btnop {
    my ($object) = @_;
    #do something...
    mpfc::WSCbase_setProperty($object,"labelString","hello");
}
```

This is the event procedure to be called when the button is activated (clicked). The notation of the method and the arguments are similar with what event procedures written in C++ use, and you may be familiar with those if you know the WideStudio. It will be described in the following sections.

3.3 Execute on command line

After the first execution with the Application builder, you can run the script on the command line in the directory in which the script files are.

For example,

```
% ruby newproject.rb
```

or

```
% python newproject.py
```

or

```
% perl newproject.pl
```

Note that you have to "Execute" or "Build all" on the Application Builder at least one time for generating the script file "**`newproject.{rb|py|pl}`**" from the project file "**`newproject.prj`**".

4 Development

Here we describe how to make an application with the scripts.

4.1 Making a new project

Select the menu bar "Project" — "New project", or "New project" icon on the tool bar.

Fill the name of your project to "Project name", and to "Working dir" the directory where files of the project are stored. Next.

Select "Project type" as "Normal application". Next.

Select "Locale type" as "English", then choose "Language" (for programming) as "Ruby" or "Python" or "Perl". Finish.

4.2 Making a new window

Select the menu bar "File" — "New window", or press the "New application window" icon.

Then, check "Type" as "Normal window". Next.

Check "Project" as "Add to project", then fill the "Name" of the window (default is newwin000). Next. Note that the name of the window should not be the same with the project name.

Select one of items as "Templates". Finish.

4.3 Putting components onto the window

The description in this subsection is almost same with the ordinal development with the Application Builder in C++. So skip here if you already know the things.

Display the object box by selecting the menu bar "View" — "ObjectBox", or press the "Display the object box" icon.

Select one of the components, for example, button class WSCvbtn, then drag it keeping pressing the left button of the mouse, and drop onto the window.

Assume that the name of the button is "newvbtn_000". Select it, and open the Procedure Editor in the Application Builder; select the "Procedures" tab at the top of the bottom-right pane. Next, click the "Add procedure" icon in the Procedure Editor.

After the CreateProcedure dialog comes up, fill the following fields:

Procedure name The name of the event procedure. Any strings including spaces and multi-bytes characters are accepted. No need to be unique.

Function name The name of the function called when the event procedure is invoked. Therefore, this should be unique in your application, and be an valid name for the script.

Trigger The event where the event procedure is invoked. Select one from the pulldown menu.

Then press “OK” button, and a file for the procedure is created. Note that the name of the file is as follows:

Ruby “Function name”+“.rb”

Python “Function name”+“.py”

Perl “Function name”+“.pl”

C++ “Function name”+“.cpp”

4.4 Coding the procedure

An event procedure is created, select it on the Procedure Editor and double-click it. Then a file with the template code for the procdure function appears with an text editor that you specified in the project setting.

For example, an event procedure with the function “changestring” produce the following file:

Ruby file: `changestring.rb`

```
#----------------------------------------------------------
#Function for the event procedure
#----------------------------------------------------------
def changestring(object)
#do something...

end
```

Write your codes between the line “`#do something...`” and “`end`”.

Python file: `changestring.py`

```
#----------------------------------------------------------
#Function for the event procedure
#----------------------------------------------------------
import mpfc

def changestring(object):
        #do something...

        return
mpfc.WSGFfunctionRegister("changestring",changestring)
```

Write your codes between the line “`#do something...`” and “`return`”. Do not delete the lines “`import mpfc`” and “`mpfc.WSGFfunctionRegister...`”.

Perl file: `changestring.pl`

```
#--------------------------------------------------------
#Function for the event procedure
#--------------------------------------------------------
use mpfc;

sub changestring {
    my ($object) = @_;
    #do something...

}
1;
```

Write your codes between the line "`#do something...`" and the brace "}". Do not delete the lines "`use mpfc`" and "`1;`".

C++ file: `changestring.cpp`

```
#include <WScom.h>
#include <WSCfunctionList.h>
#include <WSCbase.h>
//--------------------------------------------------------
//Function for the event procedure
//--------------------------------------------------------
void changestring(WSCbase* object){
  //do something...
}
static WSCfunctionRegister  op("changestring",(void*)changestring);
```

4.5 Example codes

For ruby and python, the way how to write a code is similar with what we do in C++. However, the syntax for perl is different.

For example, for changing the text in the button and font, the codes are shown as follows:

For ruby:

```
def changestring(object)
  object.setProperty("labelString", "hello, world")
  object.setProperty("font", 7)
end
```

The instance that call the procedure is passed by the argument "object". Methods are called with dot "." and the name of the method after the object.

For python:

```
def changestring(object):
  object.setProperty("labelString", "hello, world")
  object.setProperty("font", 7)
  return
```

The instance that call the procedure is passed by the argument "object". Methods are called with dot "." and the name of the method after the object.

For Perl:

```
sub changestring{
    my $object = @_;
    mpfc::WSCbase_setProperty($object, "labelString", "hello, world");
    mpfc::WSCbase_setProperty($object, "font", 7);
}
```

The instance that call the procedure is passed by the first element of the array of the arguments “`@_`”; you have to use the local variable substituted by “**`my $object = @_`**”.

Methods are called as a function whose name comprises of “mpfc::” + “name of class” + “_” + “name of method”. The object about which the method is called is the first argument of the function.

For C++:

```
void changestring(WSCbase* object){
    object->setProperty(WSNlabelString, "hello world");
    object->setProperty(WSNfont, 7);
}
```

4.6 Tips

4.6.1 Methods

Not all the methods that are availabel in C++ are implemented for Ruby and Python. This will be improved.

4.6.2 Properties

For the script, property name should be specifiled as a string: e.g., `‘‘labelString’’` or `‘‘font’’`, On the other hand, for C++, the property is used with symbols: e.g. `WSNlabelString` or `WSNfont`.

The conversion from the symbols to the strings is just simple. Remove the first three letters (WSN) and surround it by double quotations (“ ”).

4.6.3 Global functions

If you want to use a global function, you need an appropreate prefix to the name of the funtion.

Ruby prefix: “Mpfc::”

```
msg = Mpfc::WSGIappMessageDialog()
msg.setVisible(1)
```

Python prefix: “mpfc.”

```
msg = mpfc.WSGIappMessageDialog()
msg.setVisible(1)
```

Perl prefix: “mpfc::”

```
$msg = mpfc::WSGIappMessageDialog();
mpfc::WSCbase_setVisible($msg, 1);
```

C++

```
msg = WSGIappMessageDialog();
msg->setVisible(1);
```

4.6.4 Relation between instances

To refer another instance, use the global instance of WSCbaseList.

Ruby

```
obj = Mpfc::WSGIappObjectList().getInstance("WSCbase", "newvbtn_000")
```

Python

```
obj = mpfc.WSGIappObjectList().getInstance("WSCbase", "newvbtn_000")
```

Perl

```
$obj = mpfc::WSGIappObjectList().getInstance("WSCbase", "newvbtn_000");
```

C++

```
obj = WSGIappObjectList()->getInstance("WSCbase", "newvbtn_000");
```

5 Conclusions

Now the WideStudio allows us to use scripts in Ruby, Python and Perl as well as C++.

The latest version of the WideStudio is ver.3.20-1 at 2002/Dec/13, and many of feedbacks will be reflected on the future release. In Japan, the first book about the WideStudio has been published 2002/Sep/3 by ASCII Co., Japan, and the second japanese book is now written and will be published by Mainichi Communications Inc., Japan.

We hope you enjoy programming with the WideStudio.

Resources

[1] WideStudio web site
`www.widestudio.org`

[2] Sourceforge.net project
`sourceforge.net/projects/widestudio/`

[3] Mail list
`http://lists.sourceforge.net/lists/listinfo/widestudio-users`

[4] Mail list (in Japanese)
`www.egroup.co.jp/group/widestudio/`

[5] Developers' Mail list (in Japanese)
`www.egroup.co.jp/group/dev-widestudio/`

[6] SWIG
`www.swig.org`